CS422 - Programming Language Design

Operational Semantics

Grigore Rosu

Department of Computer Science
University of Illinois at Urbana-Champaign

Operational Semantics

By an *operational semantics* of a programming language, one typically understands a collection of rules specifying how its expressions, statements, programs, etc., are evaluated or executed. These rules say how a possible implementation of a programming language should "operate" and it is not difficult in practice to give an implementation of (an interpreter of) a language in any programming languages by just following and translating its operational semantics into the target implementation language.

There is no definite agreement on how an operational semantics of a language should be given, because any description of a programming language which is rigorous enough to quickly lead to a correct implementation of the language can be considered to be a valid operational semantics.

In this part of the course, we will investigate a particular but very common operational semantics approach, called *structural operational semantics* and abbreviated *SOS*, as well as two variations aiming at improving modularity and compactness of language definitions, namely *modular SOS (MSOS)* and *reduction semantics with evaluation contexts*. Language definitional frameworks or styles are typically called "structural" when their authors want to convey the intuition that programming languages are defined inductively over the structure of their syntax.

There are two common SOS definitional styles of programming languages, big-step SOS and small-step SOS. They both consist of defining deduction (or inference, or derivation) systems for binary relations on *configurations*, where a configuration is typically a tuple containing some program fragment together with state infrastructure necessary to evaluate it. Even though big-step SOS can be technically regarded as a special case of the small-step SOS

where the small-step is big enough to take a program or a fragment of it directly to its final result, the two styles tend to be regarded as conceptually rather different.

- *Big-Step SOS*, also known as *natural semantics*. Under big-step SOS, the atomic "provable" entities are relations of configurations, typically written $C \Downarrow C'$, with the meaning that C' is the configuration obtained after the (complete) evaluation of C. A big-step SOS describes in a divide-and-conquer manner how final evaluation results of language constructs can be obtained by combining the evaluation results of their syntactic counterparts (subexpressions, substatements, etc.). For example, the big-step SOS definition of addition in a language whose expression evaluation can have side effects is

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle,\ \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i, \sigma_2\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{1}$$

Here, the meaning of a relation $\langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle$ is that arithmetic expression a is evaluated in state σ to integer i and new state σ'. If expression evaluation is side-effect-free, then one can drop the state from the right-side configuration and thus write big-step relations as $\langle a, \sigma \rangle \Downarrow \langle i \rangle$; big-step SOS definitions transform as expected:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle,\ \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle i \rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{2}$$

- *Small-Step SOS*, also known as *transitional semantics*, is how SOS was originally introduced. Under small-step SOS, the atomic provable entities are "one-computation-step" transitions, showing how a fragment of program is advanced *one step* in its evaluation process; a small-step SOS identifies for each language construct typically one of its syntactic counterparts which can be advanced precisely one step, and

then shows how that sub-computation step translates into a one-computation step for the language construct. For example, the small-step SOS definition of addition is

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1' + a_2, \sigma\rangle} \tag{3}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1 + a_2', \sigma\rangle} \tag{4}$$

$$\frac{\cdot}{\langle i_1 + i_2, \sigma\rangle \rightarrow \langle i, \sigma\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{5}$$

We will be able to formally show that a step in a big-step SOS of a language corresponds to many steps in a small-step SOS of the same language.

Many recent works on SOS definitions of languages take the liberty to mix big-step and small-step. For example, one may evaluate the

condition of a conditional statement in one big-step, but then transit to the left or to the right branch of the conditional in a small-step; or, more generally, one can use big-step for expressions and small-step for statements.

Syntax of a Simple Language

We will exemplify the conventional SOS definitional styles, as well as variations of them, by means of a very simple non-procedural imperative language which has arithmetic and boolean expressions, conditionals and while loops.

A program is a sequence of statements followed by an expression. The expression is evaluated in the state obtained after evaluating all the statements and its result is returned as the result of the evaluation of the entire program. Arithmetic and boolean expressions do not have side effects; only statements can change the state. This language is reminiscent of the IMP language defined by Winskel in his book

> G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

Formally, the syntax of this simple language can be given as a context-free grammar (CFG) as follows:

$$
\begin{array}{rcl}
\mathit{Var} & ::= & \text{standard identifiers} \\
\mathit{AExp} & ::= & \mathit{Var} \mid 1 \mid 2 \mid 3 \mid \ldots \mid \\
 & & \mathit{AExp} + \mathit{AExp} \mid \mathit{AExp} - \mathit{AExp} \mid \mathit{AExp} * \mathit{AExp} \mid \mathit{AExp} / \mathit{AExp} \\
\mathit{BExp} & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathit{AExp} \leq \mathit{AExp} \mid \mathit{AExp} \geq \mathit{AExp} \mid \mathit{AExp} = \mathit{AExp} \\
 & & \mathit{BExp}\ \mathsf{and}\ \mathit{BExp} \mid \mathit{BExp}\ \mathsf{or}\ \mathit{BExp} \mid \mathsf{not}\ \mathit{BExp} \\
\mathit{Stmt} & ::= & \mathsf{skip} \mid \mathit{Var} := \mathit{AExp} \mid \mathit{Stmt}\ ;\ \mathit{Stmt} \mid \{\mathit{Stmt}\} \\
 & & \mathsf{if}\ \mathit{BExp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} \mid \mathsf{while}\ \mathit{BExp}\ \mathit{Stmt} \\
\mathit{Pgm} & ::= & \mathit{Stmt}\ ;\ \mathit{AExp}
\end{array}
$$

State, Configuration, Transition, Rule

An SOS defines a relation on *configurations*, which is typically binary and thought of as a *transition relation*. In general, a configuration is a tuple containing a program or a fragment of program, which we may call a *term over the syntax of the language*, and corresponding needed semantic infrastructure, such as a state, various control stacks, etc.; however, in our simple language definition we only need configurations consisting of pairs of a term and a state.

For this simple imperative language, a *state* is a map from variables to integer numbers $Var \to Int$. We let σ, σ', etc., denote states. If σ is a state and x a variable, then we let $\sigma[x]$ or $\sigma(x)$ denote the integer value to which σ maps x. Moreover, if x is a variable and i an integer, then we let $\sigma[x \leftarrow i]$ denote the function $Var \to Int$ defined as follows:

$$\sigma[x \leftarrow i](y) = \begin{cases} \sigma(y) & \text{if } x \neq y, \\ i & \text{if } x = y. \end{cases}$$

We let $\emptyset$ denote the initial state. We assume that states are *partial* functions, that is, a state is assumed undefined in a variable if that variable was never assigned a value in the state; in particular, $\emptyset[x]$ is undefined for any x.

In our simple language definition, we only need simple *configurations*. We enclose the various components forming a configuration with angle brackets. For example, $\langle a, \sigma \rangle$ is a configuration containing an arithmetic expression a and a state σ, and $\langle b, \sigma \rangle$ is a configuration containing a boolean expression b and a state σ. Configurations of different types need not necessarily have the same number of components. For example, since all programs evaluate in the initial state, there is no need to mention a state next to a program in a configuration; in this case, a

configuration is simply a one element tuple, $\langle p \rangle$, where p is a program. We will introduce configurations tacitly by need in our SOS language definitions; what distinguishes configurations from other structures are the angle brackets.

The core ingredient of an SOS definition is the *sequent*. Like in definitions of deductive systems in general, a sequent can be almost any tuple; however, in our particular SOS definitions, a *sequent* is typically a *transition*. We use arrows for transitions, such as $\Downarrow$ or $\rightarrow$. A transition takes a configuration to another configuration (the "next step"). For example, in a big-step SOS, a transition that signifies that arithmetic expression a evaluates in state σ to integer i can be written $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ if the evaluation of expressions is side-effect free, or $\langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle$ if the evaluation of expressions may have side effects, where σ' is the state after the evaluation of a to i in σ. Also, in a small-step semantics, a transition $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ signifies that statement s in state σ transits to

statement s' (typically a variant of s whose computation is advanced by one step) in state σ'.

An SOS definition is a collection of *parametric rules* of the form:

$$\frac{\mathit{transition}_1,\ \mathit{transition}_2,\ \ldots,\mathit{transition}_k}{\mathit{transition}}$$

The transitions above the line are called the *condition* of the rule, and the transition below the line is called the *conclusion* of the rule. The intuition here is that *transition* is possible whenever $\mathit{transition}_1$, $\mathit{transition}_2$, ..., $\mathit{transition}_k$ are possible. We may also say that *transition* is *derivable*, or can be *inferred*, from $\mathit{transition}_1$, $\mathit{transition}_2$, ..., $\mathit{transition}_k$. This reflects the fact that an SOS definition can (and should) be viewed as a logical system, where one can deduce possible behaviors of programs.

If $k = 0$, then we call the rule *unconditional* and simply write

$$\frac{\cdot}{\mathit{transition}}$$

Big-Step SOS

Introduced as *natural semantics* in

> G. Kahn. *Natural semantics.* In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, STACS, volume 247 of Lecture Notes in Computer Science, pages 22–39. Springer, 1987,

also named *relational semantics* or *evaluation semantics* in

> R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997,

big-step semantics is "the most denotational" of the operational semantics. One can view big-step definitions as definitions of functions, or more generally of relations, interpreting each language construct in an appropriate domain.

Big-Step SOS Rules for Arithmetic Expressions

We here show how to derive transitions $\langle a, \sigma \rangle \Downarrow \langle i \rangle$, stating that the arithmetic expression a evaluates/executes/transits to the integer i in state σ. Note that in the case of our simple language, the transition relation is going to be *deterministic*, in the sense that $i_1 = i_2$ whenever $\langle a, \sigma \rangle \Downarrow \langle i_1 \rangle$ and $\langle a, \sigma \rangle \Downarrow \langle i_2 \rangle$ can be deduced. However, in the context of nondeterministic/concurrent languages, $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ states that a *may possibly* evaluate to i in state σ, but it may also evaluate to other integers.

We need to inductively define the transition relation for each language construct for arithmetic expressions. Since *Var* and *Int* are syntactic subcategories of *AExp*, we start by introducing the following two SOS rules, one for variables and the other for integers:

$$\frac{\cdot}{\langle i, \sigma\rangle \Downarrow \langle i\rangle} \tag{6}$$

We next give the SOS rules for the arithmetic operations of addition, subtraction, multiplication and division. Recall that arithmetic and boolean expression do not have side effects, so the state σ does not change. However, the state needs to be carried as part of the configuration that appears to the left side of each sequent, because it may be needed if expressions contain variables, to look them up.

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{7}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 - a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i \text{ is } i_1 \text{ minus } i_2 \tag{8}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 * a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i \text{ is } i_1 \text{ times } i_2 \qquad (9)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 / a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i_2 \neq 0 \text{ and } i \text{ is the quotient of } i_1 \text{ by } i_2 \qquad (10)$$

Note that we chose not to "short-cut" the multiplication and the quotient operators when a_1 evaluates to 0. Therefore, in this case a_2 is still expected to produce a correct value in order for the rules to be applicable (e.g., a_2 cannot perform a division by 0).

Exercise 1 *Change the big-step SOS definition above so that multiplication and quotient short-cut when a_1 evaluates to 0.*

All rules above and below are *parametric*, that is, they can be viewed as collections of concrete *instance rules*. For example, a possible instance of rule (7) can be the following:

$$\frac{\langle 1, \emptyset\rangle \Downarrow \langle 1\rangle,\ \langle 2, \emptyset\rangle \Downarrow \langle 2\rangle}{\langle 1+2, \emptyset\rangle \Downarrow \langle 3\rangle}$$

Another possible instance of rule (7) can be the following (assume σ is some concrete state), which, of course, seems problematic:

$$\frac{\langle 1, \sigma\rangle \Downarrow \langle 1\rangle,\ \langle 2, \sigma\rangle \Downarrow \langle 9\rangle}{\langle 1+2, \sigma\rangle \Downarrow \langle 10\rangle}$$

The rule above is indeed a correct instance of (7). However, one will never be able to infer $\langle 2, \sigma\rangle \Downarrow \langle 9\rangle$, so this rule cannot be applied in a correct inference.

The following is a correct inference, where x and y are any variables and σ is some concrete state with $\sigma[x] = \sigma[y] = 1$:

$$\dfrac{\dfrac{}{\langle x,\sigma\rangle \Downarrow \langle 1\rangle}\,,\quad \dfrac{\dfrac{}{\langle y,\sigma\rangle \Downarrow \langle 1\rangle}\,,\ \dfrac{}{\langle x,\sigma\rangle \Downarrow \langle 1\rangle}}{\langle y * x,\sigma\rangle \Downarrow \langle 1\rangle}\,,\ \dfrac{}{\langle 2,\sigma\rangle \Downarrow \langle 2\rangle}}{\langle y * x + 2,\sigma\rangle \Downarrow \langle 3\rangle}}{\langle x - (y * x + 2),\sigma\rangle \Downarrow \langle -2\rangle}$$

The proof above can be regarded as an upside-down tree, with dots as leaves and instances of SOS rules as nodes. We call such "complete" (in the sense that their leaves are all dots and their nodes are correct instances of SOS rules) trees *proof trees*. This way, we have a way to mathematically *derive facts* about programs within their SOS semantics. We may call the root of a proof tree the *fact that was proved or derived*, and the tree *its proof or derivation*.

The conditions which make a rule applicable, such as, for example, "where i is the sum of i_1 and i_2" in rule (7), are called *side conditions*. Side conditions typically constrain variables over

recursively enumerable domains (e.g., i, i_1 and i_2 range over integer numbers in rule (7). Therefore, each SOS rule comprises a *recursively enumerable* collection of concrete instance rules. One could also move the side-conditions into the actual condition of the rule (above the line), but many rules that are unconditional would then become conditional and thus harder to type and read.

Big-Step SOS Rules for Boolean Expressions

We can now similarly add transitions of the form $\langle b, \sigma\rangle \Downarrow \langle t\rangle$, where b is a boolean expression and t is a truth value in the set $\{\mathsf{true}, \mathsf{false}\}$:

$$\frac{\cdot}{\langle \mathsf{true}, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle} \tag{11}$$

$$\frac{\cdot}{\langle \mathsf{false}, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle} \tag{12}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \leq a_2, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle} \text{ where } i_1 \text{ less than or equal to } i_2 \tag{13}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \leq a_2, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle} \text{ where } i_1 \text{ larger than } i_2 \tag{14}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \geq a_2, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle} \text{ for } i_1 \text{ larger than or equal to } i_2 \quad (15)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \geq a_2, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle} \text{ where } i_1 \text{ smaller than } i_2 \quad (16)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i\rangle}{\langle a_1 \ = \ a_2, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle} \quad (17)$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 \ = \ a_2, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle} \text{ where } i_1 \text{ different from } i_2 \quad (18)$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle,\ \langle b_2, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle}{\langle b_1 \ \mathsf{and} \ b_2, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle} \quad (19)$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle t_1\rangle,\ \langle b_2, \sigma\rangle \Downarrow \langle t_2\rangle}{\langle b_1 \ \mathsf{and} \ b_2, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle} \text{ where } t_1 \text{ or } t_2 \text{ is } \mathsf{false} \quad (20)$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle \;\; \langle b_2, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle}{\langle b_1 \;\; \mathsf{or} \;\; b_2, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle} \tag{21}$$

$$\frac{\langle b_1, \sigma\rangle \Downarrow \langle t_1\rangle, \;\; \langle b_2, \sigma\rangle \Downarrow \langle t_2\rangle}{\langle b_1 \;\; \mathsf{or} \;\; b_2, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle} \text{ where } t_1 \text{ or } t_2 \text{ is } \mathsf{true} \tag{22}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle}{\langle \mathsf{not} \;\; b, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle} \tag{23}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle}{\langle \mathsf{not} \;\; b, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle} \tag{24}$$

Note that we chose not to shortcut the boolean operators either.

Big-Step SOS Rules for Statements

Statements in our simple imperative language change the state, so we need to introduce a new transition relation of the form $\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle$, where s is a statement and σ, σ' are states. The SOS rules for statements are then:

$$\frac{\cdot}{\langle \mathsf{skip}, \sigma\rangle \Downarrow \langle \sigma\rangle} \tag{25}$$

$$\frac{\langle a, \sigma\rangle \Downarrow \langle i\rangle}{\langle x := a, \sigma\rangle \Downarrow \langle \sigma[x \leftarrow i]\rangle} \tag{26}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma''\rangle,\ \langle s_2, \sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle s_1; s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{27}$$

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \{s\}, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{28}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle,\ \langle s_1, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \mathsf{if}\ \ b\ \ \mathsf{then}\ \ s_1\ \ \mathsf{else}\ \ s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{29}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle,\ \langle s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \mathsf{if}\ \ b\ \ \mathsf{then}\ \ s_1\ \ \mathsf{else}\ \ s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{30}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathsf{false}\rangle}{\langle \mathsf{while}\ \ b\ s, \sigma\rangle \Downarrow \langle \sigma\rangle} \tag{31}$$

$$\frac{\langle b, \sigma\rangle \Downarrow \langle \mathsf{true}\rangle,\ \langle s, \sigma\rangle \Downarrow \langle \sigma''\rangle,\ \langle \mathsf{while}\ \ b\ s, \sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle \mathsf{while}\ \ b\ s, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{32}$$

Big-Step SOS Rules for Programs

Programs are always executed in the initial state, so we can define their SOS rule as follows:

$$\frac{\langle s, \emptyset \rangle \Downarrow \langle \sigma \rangle,\ \langle a, \sigma \rangle \Downarrow \langle i \rangle}{\langle s; a \rangle \Downarrow \langle i \rangle} \tag{33}$$

On Enumerability and Termination

Since each SOS rule comprises a *recursively enumerable* collection of concrete instance rules, and since a language definition consists of a finite set of ("parametric") SOS rules, by enumerating all the concrete instances of these rules we get a recursively enumerable set of concrete instance rules.

Furthermore, since proof trees built with nodes in a recursively

enumerable set are themselves recursively enumerable, it follows that the set of proof trees is recursively enumerable. In other words, we can find an algorithm that lists all the proof trees, in particular one that enumerates all the derivable transitions $\langle p \rangle \Downarrow \langle i \rangle$ for all programs p that "evaluate" to i. Note, however, that we only informally know the meaning of "evaluate" in the previous sentence. Formally, we say *by definition* that

- Program p *evaluates to* i iff the transition $\langle p \rangle \Downarrow \langle i \rangle$ is derivable, that is, iff there is a proof tree whose root is $\langle p \rangle \Downarrow \langle i \rangle$.
- Program p *(correctly) terminates* iff there exists some integer i such that $\langle p \rangle \Downarrow \langle i \rangle$ is derivable.

Intuitively, a program may also terminate when it performs a division by zero. However, we cannot capture such erroneous termination with our current big-step semantics. We would need to add a special "error" value and propagate it through all the

language constructs.

Exercise 2 *Add an* error *value and modify the big-step semantics to allow derivations of the form* $\langle p \rangle \Downarrow \langle \mathsf{error} \rangle$ *when* p *"performs" a division by zero.*

Exercise 3 *Prove that the transition relation defined above is* deterministic*, that is:*

- *If* $\langle a, \sigma \rangle \Downarrow \langle i \rangle$ *and* $\langle a, \sigma \rangle \Downarrow \langle i' \rangle$ *derivable then* $i = i'$*;*
- *If* $\langle b, \sigma \rangle \Downarrow \langle t \rangle$ *and* $\langle b, \sigma \rangle \Downarrow \langle t' \rangle$ *derivable then* $t = t'$*;*
- *If a program* p *terminates then there is a unique* i *such that* $\langle p \rangle \Downarrow \langle i \rangle$ *is derivable.*

The above holds because our simple programming language is deterministic; we will see later in the class that concurrent programming languages may manifest non-deterministic behaviors (for example, due to data-races).

By enumerating all proof trees, one can eventually find such an i for any terminating program p. This simple-minded algorithm may take a very long time and a huge amount of resources, but it is theoretically important to understand that it can be done.

Exercise 4 *Show that there is no algorithm which takes as input a program p and says whether it terminates or not.*

The above holds because our simple language, due to its while loops, is Turing-complete. Thus, if one was able to decide termination of programs in our language then one was able to also decide termination of Turing machines, which would contradict one of the basic undecidable problems, namely the *halting problem*.

An interesting observation here is that non-termination of a program corresponds to *lack of proof*, and that the latter is not decidable in many interesting logics. Indeed, in *complete* logics, that is logics that admit a complete proof system, one can

enumerate all the truths. However, in general there is not much one can do about non-truths, because the enumeration algorithm will loop forever when run on a non-truth. In decidable logics one can enumerate both truths and non-truths; clearly, decidable logics are not powerful enough for our task of defining programming languages, exactly because of the halting problem argument above.

Small-Step SOS

Introduced by Plotkin in

> G. D. Plotkin. *A structural approach to operational semantics*. Journal of Logic and Algebraic Programming, 60-61:17–139, 2004. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

also called *transition semantics* or *reduction semantics*, small-step semantics captures the notion of one atomic computational step.

Small-Step SOS Rules for Arithmetic Expressions

Unlike in a big-step SOS definition where one defines many (all) computation steps in one transition, in a small-step SOS definition a transition encodes only one step of computation. To distinguish small-step transitions from big-step ones, we use a plain arrow $\rightarrow$ instead of $\Downarrow$. The next rule happens to be almost the same as in the big-step SOS semantics; that's because variable lookup is an atomic-step operation both in big-step and in small-step semantics:

$$\frac{}{\langle x, \sigma \rangle \rightarrow \langle \sigma[x], \sigma \rangle} \text{ if } \sigma[x] \text{ defined} \tag{34}$$

To avoid treating special cases in the subsequent small-step definitions, we prefer to mention the state in the right-side configuration even if it does not change.

Exercise 5 *Suppose that one wants to drop the state from all the right-side configurations of all the rules that do not change the state, including the one above. Is that possible for all the rules? Rewrite the small-step SOS definition of our simple language maintaining minimal configurations in the right sides of transitions. What is the drawback of this approach?*

Recall that in big-step SOS we had a rule for evaluating constant (integer) arithmetic expressions to their corresponding value, namely rule (6). Do we need a corresponding small-step SOS rule for evaluating an integer-number expression into its corresponding value, or, in other words, do we need a small-step SOS rule of the form below?

$$\frac{\cdot}{\langle i, \sigma \rangle \rightarrow \langle i, \sigma \rangle} \tag{35}$$

The answer is "no", because an integer-number expression is already a value in our language, so no reductions are further

desired on it. Moreover, adding such a rule would lead to undesired divergent SOS reductions later on when we consider the transitive closure of the one-step relation $\rightarrow$.

We next give the small-step SOS rules for the arithmetic operations of addition, subtraction, multiplication and division:

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1' + a_2, \sigma\rangle} \tag{36}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1 + a_2', \sigma\rangle} \tag{37}$$

$$\frac{\cdot}{\langle i_1 + i_2, \sigma\rangle \rightarrow \langle i, \sigma\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{38}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 - a_2, \sigma\rangle \rightarrow \langle a_1' - a_2, \sigma\rangle} \tag{39}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 - a_2, \sigma\rangle \to \langle a_1 - a_2', \sigma\rangle} \tag{40}$$

$$\frac{\cdot}{\langle i_1 - i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i \text{ is } i_1 \text{ minus } i_2 \tag{41}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 * a_2, \sigma\rangle \to \langle a_1' * a_2, \sigma\rangle} \tag{42}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 * a_2, \sigma\rangle \to \langle a_1 * a_2', \sigma\rangle} \tag{43}$$

$$\frac{\cdot}{\langle i_1 * i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i \text{ is the product of } i_1 \text{ and } i_2 \tag{44}$$

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 / a_2, \sigma\rangle \to \langle a_1' / a_2, \sigma\rangle} \tag{45}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1/a_2, \sigma\rangle \to \langle a_1/a_2', \sigma\rangle} \tag{46}$$

$$\frac{\cdot}{\langle i_1/i_2, \sigma\rangle \to \langle i, \sigma\rangle} \text{ where } i_2 \neq 0 \text{ and } i \text{ is the quotient of } i_1 \text{ by } i_2 \tag{47}$$

As in the case of "big step" rules, the "small step" SOS rules are also parametric. The following is a correct inference, where x and y are any variables and σ is any state with $\sigma[x] = 1$:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cdot}{\langle x, \sigma\rangle \to \langle 1, \sigma\rangle}}{\langle y * x, \sigma\rangle \to \langle y * 1, \sigma\rangle}}{\langle y * x + 2, \sigma\rangle \to \langle y * 1 + 2, \sigma\rangle}}{\langle x - (y * x + 2), \sigma\rangle \to \langle x - (y * 1 + 2), \sigma\rangle}$$

The above can also be regarded as a proof, but this time of the fact

that replacing the second occurrence of x by 1 is a correct one-step computation.

Small-Step SOS Rules for Boolean Expressions

We can now similarly add transitions of the form $\langle b, \sigma \rangle \to \langle b', \sigma \rangle$, where b is a reducible boolean expression and σ is a state:

$$\frac{\langle a_1, \sigma \rangle \to \langle a_1', \sigma \rangle}{\langle a_1 \leq a_2, \sigma \rangle \to \langle a_1' \leq a_2, \sigma \rangle} \tag{48}$$

$$\frac{\langle a_2, \sigma \rangle \to \langle a_2', \sigma \rangle}{\langle a_1 \leq a_2, \sigma \rangle \to \langle a_1 \leq a_2', \sigma \rangle} \tag{49}$$

$$\frac{\cdot}{\langle i_1 \leq i_2, \sigma \rangle \to \langle \mathsf{true}, \sigma \rangle} \text{ where } i_1 \text{ smaller than or equal to } i_2 \tag{50}$$

$$\frac{\cdot}{\langle i_1 \leq i_2, \sigma\rangle \rightarrow \langle \mathsf{false}, \sigma\rangle} \text{ where } i_1 \text{ larger than } i_2 \tag{51}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 \geq a_2, \sigma\rangle \rightarrow \langle a_1' \geq a_2, \sigma\rangle} \tag{52}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 \geq a_2, \sigma\rangle \rightarrow \langle a_1 \geq a_2', \sigma\rangle} \tag{53}$$

$$\frac{\cdot}{\langle i_1 \geq i_2, \sigma\rangle \rightarrow \langle \mathsf{true}, \sigma\rangle} \text{ where } i_1 \text{ larger than or equal to } i_2 \tag{54}$$

$$\frac{\cdot}{\langle i_1 \geq i_2, \sigma\rangle \rightarrow \langle \mathsf{false}, \sigma\rangle} \text{ where } i_1 \text{ smaller than } i_2 \tag{55}$$

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 \ = \ a_2, \sigma\rangle \rightarrow \langle a_1' \ = \ a_2, \sigma\rangle} \tag{56}$$

$$\frac{\langle a_2, \sigma\rangle \to \langle a_2', \sigma\rangle}{\langle a_1 \ = \ a_2, \sigma\rangle \to \langle a_1 \ = \ a_2', \sigma\rangle} \tag{57}$$

$$\frac{\cdot}{\langle i \ = \ i, \sigma\rangle \to \langle \mathsf{true}, \sigma\rangle} \tag{58}$$

$$\frac{\cdot}{\langle i_1 \ = \ i_2, \sigma\rangle \to \langle \mathsf{false}, \sigma\rangle} \text{ where } i_1 \neq i_2 \tag{59}$$

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \ \mathsf{and} \ b_2, \sigma\rangle \to \langle b_1' \ \mathsf{and} \ b_2, \sigma\rangle} \tag{60}$$

$$\frac{\langle b_2, \sigma\rangle \to \langle b_2', \sigma\rangle}{\langle b_1 \ \mathsf{and} \ b_2, \sigma\rangle \to \langle b_1 \ \mathsf{and} \ b_2', \sigma\rangle} \tag{61}$$

$$\frac{\cdot}{\langle t_1 \ \mathsf{and} \ t_2, \sigma\rangle \to \langle t, \sigma\rangle} \text{ where } t_1,\ t_2,\ t \text{ are truth values s.t. } t \text{ is ``}t_1 \text{ and } t_2\text{''} \tag{62}$$

Note that the rule above has 4 instances wrt t_1, t_2 and t, each further parametric in σ; however, the state σ cannot influence the applicability of these 4 instance rules.

$$\frac{\langle b_1, \sigma\rangle \to \langle b_1', \sigma\rangle}{\langle b_1 \ \mathsf{or} \ b_2, \sigma\rangle \to \langle b_1' \ \mathsf{or} \ b_2, \sigma\rangle} \tag{63}$$

$$\frac{\langle b_2, \sigma\rangle \to \langle b_2', \sigma\rangle}{\langle b_1 \ \mathsf{or} \ b_2, \sigma\rangle \to \langle b_1 \ \mathsf{or} \ b_2', \sigma\rangle} \tag{64}$$

$$\frac{\cdot}{\langle t_1 \ \mathsf{or} \ t_2, \sigma\rangle \to \langle t, \sigma\rangle} \text{ where } t_1,\ t_2,\ t \text{ are truth values s.t. } t \text{ is ``}t_1 \text{ or } t_2\text{''} \tag{65}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \mathsf{not} \ b, \sigma\rangle \to \langle \mathsf{not} \ b', \sigma\rangle} \tag{66}$$

$$\frac{\cdot}{\langle \mathsf{not}\ \ t, \sigma\rangle \rightarrow \langle t', \sigma\rangle} \text{ where } t \text{ and } t' \text{ are opposite truth values} \quad (67)$$

Small-Step SOS Rules for Statements

There are two cases to distinguish in the small-step SOS rules for statements: the defined step is an intermediate step, or the defined step is the final step in the evaluation of the statement. These cases may suggest two distinct approaches to giving small-step SOS rules for statements, both of them discussed in the sequel. One approach is to use transitions of the form $\langle s, \sigma\rangle \rightarrow \langle s', \sigma'\rangle$ when the statement s needs further processing and to use transitions of the form $\langle s, \sigma\rangle \rightarrow \langle \sigma'\rangle$ when the statement s needs no further processing, such as in the case of skip. Another approach is to regard skip as a "statement value" that any other (terminating)

statement "evaluates to", and then to use only transitions of the form $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$, with no such transition when s is skip. Since SOS is a *syntactic* language definitional technique, in the sense that it builds upon the belief that program evaluation can be carried over as a sequence of syntactic transformations of the original program, the second approach may have to slightly disobey this belief if the language to define has no skip statement, by introducing an artificial one inexistent in the original syntax.

Variant 1

Let us assume two additional types of configurations, namely $\langle s, \sigma \rangle$ and $\langle \sigma \rangle$, for statements s and states σ. The following small-step SOS rules involve both transitions of the form $\langle s, \sigma \rangle \rightarrow \langle s', \sigma' \rangle$ (when s needs further processing) and of the form $\langle s, \sigma \rangle \rightarrow \langle \sigma' \rangle$ (when s needs no further processing).

$$\frac{\cdot}{\langle \mathsf{skip}, \sigma \rangle \to \langle \sigma \rangle} \tag{68}$$

$$\frac{\langle a, \sigma \rangle \to \langle a', \sigma \rangle}{\langle x := a, \sigma \rangle \to \langle x := a', \sigma \rangle} \tag{69}$$

$$\frac{\cdot}{\langle x := i, \sigma \rangle \to \langle \sigma[x \leftarrow i] \rangle} \tag{70}$$

$$\frac{\langle s_1, \sigma \rangle \to \langle s_1', \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \to \langle s_1'; s_2, \sigma' \rangle} \tag{71}$$

$$\frac{\langle s_1, \sigma \rangle \to \langle \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \to \langle s_2, \sigma' \rangle} \tag{72}$$

In each of the two small-step SOS variants, there are two ways to give semantics to the block statement: either delaying the elimination of the block as much as possible, such as

$$\frac{\langle s, \sigma\rangle \to \langle s', \sigma'\rangle}{\langle \{s\}, \sigma\rangle \to \langle \{s'\}, \sigma'\rangle} \tag{73}$$

$$\frac{\langle s, \sigma\rangle \to \langle \sigma'\rangle}{\langle \{s\}, \sigma\rangle \to \langle \sigma'\rangle} \tag{74}$$

or eliminating the block right away:

$$\frac{\cdot}{\langle \{s\}, \sigma\rangle \to \langle s, \sigma\rangle} \tag{75}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \to \langle \mathsf{if}\ b'\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \sigma\rangle} \tag{76}$$

$$\frac{\cdot}{\langle \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \to \langle s_1, \sigma\rangle} \tag{77}$$

$$\frac{\cdot}{\langle \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, \sigma\rangle \to \langle s_2, \sigma\rangle} \tag{78}$$

$$\frac{\cdot}{\langle \mathsf{while}\ b\ s, \sigma\rangle \to \langle \mathsf{if}\ b\ \mathsf{then}\ (s;\ \mathsf{while}\ b\ s)\ \mathsf{else}\ \mathsf{skip}, \sigma\rangle} \tag{79}$$

Variant 2

Let us alternatively assume only one additional type of configuration, namely $\langle s, \sigma\rangle$, for statements s and states σ. The following small-step SOS rules involve only transitions of the form $\langle s, \sigma\rangle \to \langle s', \sigma'\rangle$.

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle x := a, \sigma\rangle \to \langle x := a', \sigma\rangle} \tag{80}$$

$$\frac{\cdot}{\langle x := i, \sigma\rangle \to \langle \mathsf{skip}, \sigma[x \leftarrow i]\rangle} \tag{81}$$

$$\frac{\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle}{\langle s_1; s_2, \sigma\rangle \to \langle s_1'; s_2, \sigma'\rangle} \tag{82}$$

$$\frac{\cdot}{\langle \mathsf{skip}; s_2, \sigma\rangle \to \langle s_2, \sigma\rangle} \tag{83}$$

If blocks are desired to be eliminated as soon as encountered then we can use the same rule as in Variant 1, namely

$$\frac{\cdot}{\langle \{s\}, \sigma\rangle \to \langle s, \sigma\rangle} \tag{84}$$

while if one wishes to delay the elimination of blocks then one needs the following two rules instead:

$$\frac{\langle s, \sigma\rangle \to \langle s', \sigma'\rangle}{\langle \{s\}, \sigma\rangle \to \langle \{s'\}, \sigma'\rangle} \tag{85}$$

$$\frac{\cdot}{\langle \{\mathsf{skip}\}, \sigma\rangle \to \langle \mathsf{skip}, \sigma\rangle} \tag{86}$$

$$\frac{\langle b, \sigma\rangle \to \langle b', \sigma\rangle}{\langle \textsf{if } b \textsf{ then } s_1 \textsf{ else } s_2, \sigma\rangle \to \langle \textsf{if } b' \textsf{ then } s_1 \textsf{ else } s_2, \sigma\rangle} \tag{87}$$

$$\frac{\cdot}{\langle \textsf{if true then } s_1 \textsf{ else } s_2, \sigma\rangle \to \langle s_1, \sigma\rangle} \tag{88}$$

$$\frac{\cdot}{\langle \textsf{if false then } s_1 \textsf{ else } s_2, \sigma\rangle \to \langle s_2, \sigma\rangle} \tag{89}$$

$$\frac{\cdot}{\langle \textsf{while } b\ s, \sigma\rangle \to \langle \textsf{if } b \textsf{ then } (s; \textsf{ while } b\ s) \textsf{ else skip}, \sigma\rangle} \tag{90}$$

Exercise 6 *For any of the two small-step variants above, can we conclude that* $\sigma = \sigma'$ *from the fact that* $\langle s, \sigma\rangle \to \langle s', \sigma'\rangle$ *is derivable?*

Exercise 7 *We have seen that the transition relation defined by our previous big-step SOS semantics was deterministic. Show that the transition relations defined by the small-step SOS semantics above are* not *deterministic. How can we change the two small-step*

SOS definitions above so that the defined transition relations become deterministic, respectively?

We will see shortly that the non-deterministic nature of the small-step transition relations does not affect the overall determinism of our language.

Small-Step SOS Rules for Programs

Programs are always executed in the initial state, so we can define their SOS rule as follows:

$$\frac{\cdot}{\langle s; a\rangle \rightarrow \langle s; a, \emptyset\rangle} \tag{91}$$

$$\frac{\langle s, \sigma\rangle \rightarrow \langle s', \sigma'\rangle}{\langle s; a, \sigma\rangle \rightarrow \langle s'; a, \sigma'\rangle} \tag{92}$$

Following the first variant of small-step SOS rules for statements we add the rule

$$\frac{\langle s, \sigma\rangle \rightarrow \langle \sigma'\rangle}{\langle s; a, \sigma\rangle \rightarrow \langle a, \sigma'\rangle} \tag{93}$$

while with the second variant we add the rule

$$\frac{\cdot}{\langle \mathsf{skip}; a, \sigma\rangle \to \langle a, \sigma\rangle} \tag{94}$$

It is interesting to note that one has some flexibility in how to give an SOS semantics to a language. The same holds true for almost any language definitional style, not only for SOS.

Exercise 8 *Show that the two small-step SOS variants above are equivalent on arithmetic and boolean expressions as well as on programs, in the sense that they define precisely the same small-step relation, but not on statements. How can that be possible?*

Relating Big-Step and Small-Step SOS Definitions

Intuitively, a big-step transition is a sequence of small-step transitions. To formally capture the notion of "sequence of transitions", we define the following *complete transitive closure* of the small-step transition, written $\mathit{Config} \rightarrow^{+} \langle i \rangle$, where *Config* is a configuration and i is an integer value:

$$\frac{\mathit{Config} \rightarrow \langle i, \sigma \rangle}{\mathit{Config} \rightarrow^{+} \langle i \rangle} \tag{95}$$

$$\frac{\mathit{Config} \rightarrow \mathit{Config}',\ \mathit{Config}' \rightarrow^{+} \langle i \rangle}{\mathit{Config} \rightarrow^{+} \langle i \rangle} \tag{96}$$

Exercise 9 *Show that if* $\mathit{Config} \rightarrow^{+} \langle i \rangle$ *is derivable for some integer* i *and some configuration Config, then Config is either a program configuration or an expression configuration.*

We can now formally state and prove the following theorem:

Theorem. *For any program* p *and any integer* i, $\langle p \rangle \Downarrow \langle i \rangle$ *derivable iff* $\langle p \rangle \rightarrow^{+} \langle i \rangle$ *derivable.*

Exercise 10 *Prove the theorem above.*

Exercise 11 *Elaborate on how non-termination is reflected in a small-step SOS semantics.*

Big-Step vs. Small-Step: Advantages/Disadvantages

The theorem above tells us that the two styles of semantics achieve eventually the same objective, at least for the simple language that we considered: they tell formally when a program p evaluates to an integer i. So when do we choose one versus the other when we define a programming language? To answer this question, one needs to reflect on and to understand the advantages and the disadvantages of each of the two definitional styles.

Advantages of Big-Step SOS.

- When it can be given to a language, it is *easy to understand* since it relates syntactic entities directly to their expected results.

- Because of its recursive nature, big-step SOS gives a strong intuition on how to *implement an interpreter* for a language. Many proponents of big-step SOS regard it "almost as an interpreter" for

the defined language.

- *More abstract*, more mathematical/denotational; therefore, one can more easily define and *prove properties* about programs; of course, because of the equivalence of the two semantics, one can also use the transitive closure of the small-step SOS transition relation, but this may be less natural than using the big-step semantics. For example, try to prove the following using both big-step and small-step semantics:

Exercise 12 *Formulate using big-step and small-step SOS, respectively, what it means for the statements* while b (while b s) *and* while b s *to be "equivalent", and then prove it.*

- Particularly useful when defining *type systems* of programming languages; there, values are replaced by their types.

Disadvantages of Big-Step SOS.

- As given, big-step SOS is *not executable*: one always needs to provide a result value or state to each language construct and then use the SOS rules to "check" whether that result value or state is indeed correct. Even though in practice one can often implement an interpreter relatively easily by disseminating the big-step SOS rules of a language, they are in fact intended to give a formal description of what is possible in a language, not to be executable "per se".

- Because of the above, big-step SOS *avoids or hides non-termination* of programs by avoiding or hiding entirely the means by which one can state that a program does not terminate. Non-termination appears as "lack of proof" in the SOS system, but so does a program that performs a division by zero (or, in more complex languages, one that produces any runtime error); it is fair to say that, in principle, "runtime errors" can be handled in big-step SOS by generating and propagating "error values".

- Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred.

- Most importantly, it is widely accepted that it is *very inconvenient or impossible to define nontrivial concurrent languages* using big-step SOS. It is fair to say that big-step SOS provides some limited support for handling non-determinism. For example, one can add a boolean constant randBool that non-deterministically evaluates to either true or false as follows:

$$\frac{\cdot}{\langle \mathsf{randBool}, \sigma \rangle \Downarrow \langle \mathsf{true} \rangle} \tag{97}$$

$$\frac{\cdot}{\langle \mathsf{randBool}, \sigma \rangle \Downarrow \langle \mathsf{false} \rangle} \tag{98}$$

The big-step SOS definition of our language extended with the

rules above can now relate programs with any of the integer values that they non-deterministically evaluate to.

Unfortunately, the situation is more complex when the non-determinism is a consequence of parallelism. Consider, for example, the addition of a language construct $s_1||s_2$ for evaluating statements s_1 and s_2 in parallel. If one naively adds rules like

$$\frac{\langle s_1, \sigma\rangle \Downarrow \langle \sigma''\rangle,\ \langle s_2, \sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle s_1||s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{99}$$

$$\frac{\langle s_2, \sigma\rangle \Downarrow \langle \sigma''\rangle,\ \langle s_1, \sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle s_1||s_2, \sigma\rangle \Downarrow \langle \sigma'\rangle} \tag{100}$$

then one fails to capture the intended non-determinism in our parallel language because one simply sequentializes the executions of the two statements instead of executing them in parallel.

Exercise 13 *Give an example of a parallel program using the*

parallel statement construct above for which a big-step SOS definition like above would miss some possible results.

To correctly and completely capture the desired non-determinism of this parallel language, one needs to monolithically detect the first actions that the parallel program can potentially take, then, for a given such action, to perform it collecting the new global state after its execution, and then to generate a big-step configuration comprising a new parallel program. Complications are mostly due to the fact that a big-step semantics for this concurrent language, in case any can be given by some hard dying proponent of big-step SOS, would not be compositional. For example, there is no way to fully characterize the derivability of $\langle (s_1; s_1') || s_2, \sigma \rangle \Downarrow \langle \sigma' \rangle$ in terms of the derivability of any group of transitions involving only the statements s_1, s_1' and s_2, such as $\langle s_1 || s_2, \sigma \rangle \Downarrow \langle \sigma'' \rangle$, etc.

Advantages of Small-Step SOS.

- It is *executable*. Indeed, one can start with a program to evaluate and keep applying SOS rules whose left configuration *matches*; to apply a small-step rule, one typically needs to call recursively the "execution" procedure on smaller fragments (the ones above the lines). Nevertheless, one can log all the successful applications of rules obtained this way and thus get an execution of the original program. Therefore, it is *easy to trace and debug*. It is relatively straightforward to implement an interpreter for a language by just following, almost blindly, a small-step SOS definition of that language.

- Thanks to the above, non-termination of programs results in non-termination of searching for a proof; however, programs that perform division by zero (or runtime errors in general) can still be evaluated step-by-step until the actual error takes place; then one can be given a *meaningful error message*, including the entire stuck

program.

- It *supports definitions of non-deterministic and/or parallel languages*, obeying the interleaving semantics approach to concurrency. In fact, our small-step SOS definition of the language above was non-deterministic; it just "happened" that the final results of evaluating expressions or programs were deterministic (but one needs to prove it). Considering the parallel statement construct $s_1||s_2$ discussed in the context of big-step SOS above, one can easily give it a small-step SOS semantics as follows:

$$\frac{\langle s_1, \sigma\rangle \rightarrow \langle s_1', \sigma'\rangle}{\langle s_1||s_2, \sigma\rangle \rightarrow \langle s_1'||s_2, \sigma'\rangle} \tag{101}$$

$$\frac{\langle s_2, \sigma\rangle \rightarrow \langle s_2', \sigma'\rangle}{\langle s_1||s_2, \sigma\rangle \rightarrow \langle s_1||s_2', \sigma'\rangle} \tag{102}$$

Disadvantages of Small-Step SOS.

- It is somehow *too low level and explicit*, which results into relatively large small-step language definitions; in particular, one needs to explicitly give all the "congruence" rules for ordinary operators such as addition, multiplication, etc.

- Small-step SOS gives only an *interleaving semantics* to a concurrent language, that is, it regards executions or behaviors of concurrent programs as linear lists of actions. Small-step SOS is inherently limited with regards to giving concurrent systems a *true concurrency semantics*, because it enforces reduction to occur only at the top, thus disallowing concurrent applications of rules. However, on parallel architectures, executions of concurrent programs are *not* interleaved; for example, statements that have only local effect in different processes can be and typically are

safely executed concurrently.

- *Less suitable for proving properties* about programs; however, if one can also give a big-step semantics of the same language and prove it equivalent to the small-step semantics, then one can perform proofs using the big-step one.

Disadvantages of both Big-Step and Small-Step SOS

(1) They are *not modular*, in the sense that the addition of new statements to a language may require one to change many other rules corresponding to definitions of unrelated statements.

Homework Exercise 1 ***(20 points)*** *Let us extend our sequential language with two common features, namely* variable increment*, which increments the variable (right away) and returns its new value in context, and* halt*, which takes an arithmetic expression, evaluates it to an integer and then stops the program with that value returned as final result:*

$$\begin{array}{rcl} AExp & ::= & \ldots \mid \texttt{++}Var \\ Stmt & ::= & \ldots \mid \textsf{halt}\ \ AExp \end{array}$$

1. ***(4 points)*** *Add variable increment to the big-step SOS: list all*

the new rules, as well as the new versions of all the already existing rules that need to change;

2. ***(2 points)*** *Do the same for small-step SOS;*
3. ***(4 points)*** *Add halt to the original big-step SOS definition (forget the increment for the time being): list all the new rules, as well as the new versions of all the already existing rules that need to change;*
4. ***(4 points)*** *Do the same for small-step SOS;*
5. ***(4 points)*** *Add both variable increment and halt to the original big-step and small-step SOS definitions: for each of them list all the new rules, as well as the new versions of all the already existing rules that need to change;*

For small-step SOS, pick one of the two variants discussed in the lecture notes.

(2 points) *Comment on why we had to make the various changes and say which of big-step and small-step SOS semantics appears to you to be more modular and why.*

(2) *Big-step and small-step SOS do not properly capture the intended computational granularity as deduction steps.* The original motivation of SOS semantics, both big-step and small-step, was to offer a mathematical, logical framework in which the otherwise purely intuitive notion of "computation" in a programming languages becomes logical deduction, or inference, in the formal SOS definition of the corresponding programming language. While one can argue that SOS proofs properly capture the final result of program computations, they do not properly capture the intended computational granularity of the defined language features.

For example, the intended computational granularity of a "halt a" statement is "evaluate a and then stop the execution of the

program with that value"; compare that with "evaluate a, then propagate its value all the way back to the top over the syntax of the executing program, and then stop the program with that value", which is what happens in both big-step and small-step SOS.

A less important violation of computation granularity appears also in the small-step SOS rule for loops: a while loop transits in one-step into a conditional statement. That wasted step was not originally intended; it is just an artifact of the fact that everything in SOS needs to be a rule. What one would like to say is that that unrolling of while loops is nothing but a structural identity, or an unobservable transition, with no intended computational content.

(3) Somehow related to the previous two disadvantages, it is in general *hard or impossible to define control-intensive language features* in SOS. For example, if one adds functions to our language together with a return statement, then one either needs to traverse back the syntax when a value is returned until the corresponding

function call is encountered, thus violating again the desired computational granularity of return, or otherwise needs to add a function stack to the configurations. In the latter case, not only that all the configurations will change having to include the stack, so all the existing rules will have to change thus breaking the modularity of the definition, but more importantly, one needs to grab the current "evaluation context" when a function is invoked in order to push it in the stack, so one can recover it when the function returns. Unfortunately, "evaluation contexts" are captured by "proof contexts" in standard SOS (we will shortly see that Felleisen's reduction semantics with evaluation contexts will make contexts explicit), and proof contexts are not first class objects in SOS derivations that can be stored and retrieved at will. The list of "SOS-inconvenient" features that can be added to the language can

of course continue: exceptions, break/continue in loops, etc.

(4) *Neither big-step nor small-step SOS provides an appropriate semantical foundation for concurrent languages*. Big-step SOS can hardly be used to define any meaningful concurrent language, while small-step SOS gives only an interleaving semantics.

(5) They are both operational and syntax-driven, so they tell us close to *nothing about models* of languages. Models, which in SOS are considered "something else", are related to both denotational semantics and realizations of languages, including implementations for them.

Modular Structural Operational Semantics (MSOS)

MSOS was introduced in

> P.D. Mosses. *Foundations of Modular SOS (extended abstract).* MFCS'99, Lecture Notes in Computer Science, Volume 1672, pages 70-80, 1999.

and then discussed in detail in

> P.D. Mosses. *Modular structural operational semantics.* Journal of Logic and Algebraic Programming, Volume 60-61:195–228, 2004.

to deal with the non-modularity issues of small-step and big-step SOS. The solution proposed in MSOS involves moving the non-syntactic state components to the labels on transitions (as provided by SOS), plus a discipline of only selecting needed

attributes from the states. There are both big-step and small-step variants of MSOS, but we discuss only small-step MSOS here.

Before we get into the technicalities of MSOS, one natural question to address is why we need modularity of language definitions. One may argue that defining a programming language is a major initiative that is being done once and for all, so having to go through the defining rules many times is, after all, not such a bad idea, because it gives one the chance to find and fix potential errors in them. Here are several reasons why modularity is desirable in language definitions:

- Having to modify many or all rules whenever a new rule is added that modifies the structure of the configuration is actually more error prone than it may seem, because rules become heavier to read and debug; for example, one can write σ instead of σ' in a right-hand-side of a rule and a different or wrong language is being defined.

- When designing a new language, as opposed to an existing well-understood language, one needs to experiment with features and combinations of features; having to do lots of unrelated changes whenever a new feature is added to or removed from the language burdens the language designer with boring tasks taking considerable time that could have been otherwise spent on actual interesting language design issues.
- There is a plethora of domain-specific languages these days, generated by the need to abstract away from low-level programming language details to important, domain-specific aspects of the application. Therefore, there is a need for rapid language design and experimentation for various domainst. Moreover, domain-specific languages tend to be very dynamic, being added or removed features frequantly as the domain knowledge evolves. It would be very nice to have the possibility to "drag-and-drop" language features in one's language, such

as functions, exceptions, objects, etc.; however, in order for that to be possible, modularity of language feature definitions is crucial.

Whether MSOS gives us such a desired framework for modular language design is still open and debatable, but it is certainly the first framework explicitly aiming at modular language design.

A transition in MSOS is of the form

$$P \xrightarrow{\mathcal{X}} P',$$

where P and P' are programs or fragments of programs and $\mathcal{X}$ is a *label describing the structure of the remaining configuration both before and after the transition*. If $\mathcal{X}$ is missing, then the remaining part of the configuration is assumed to stay unchanged.
Specifically, $\mathcal{X}$ is a record containing fields denoting the semantic components of the configuration; the preferred notation in MSOS for saying that in label $\mathcal{X}$ the semantic component associated to

the field name σ (e.g., a state or a store name) is σ_0 (e.g., a function associating values to variables) is $\mathcal{X} = \{\sigma = \sigma_0, ...\}$.

Modularity in MSOS is achieved by the record comprehension notation "..." which indicates that more fields could follow but that they are not of interest for this transition, in particular that they stay unchanged after the application of the transition. If record comprehension is used in both the condition and the conclusion of an MSOS rule, then all the occurrences of "..." stand for the same fields with the same semantics components. Fields of a label can fall in one of the following categories: *read-only*, *read-write* and *write-only*.

Read-only fields are only inspected by the rule, but not modified. For example, when reading the location of a variable in an environment, the environment is not modified.

Read-write fields come in pairs, having the same field name, except

that the "write" field name is primed. They are used for transitions modifying existing state fields. For example, a state field σ can be read and written, as illustrated by the MSOS rule for assignment:

$$x := i \xrightarrow{\{\sigma{=}\sigma_0, \sigma'{=}\sigma_0[x \leftarrow i], \ldots\}} \mathsf{skip}$$

The above rule says that, if before the transition the store was σ_0, after the transition it will become $\sigma_0[x \leftarrow i]$, updating x by i. Above, and from here on, we adopt the common convention that unconditional rules are written as just their conclusion transition, omitting their empty condition.

Write-only fields are used to record things not analyzable during the execution of the program, such as the output or the trace. Their names are always primed and they have a free monoid semantics – everything written on them is actually added at the end. A good example of the usage of write-only fields would be a

rule for defining a print language construct:

$$\mathsf{print}(i) \xrightarrow{\{out'=i,\ldots\}} \mathsf{skip}$$

where "()" stands for monoid's unit.

We next give the MSOS variant of the small-step SOS definition in the previous section. Note that we use a slightly simplified MSOS here. In MSOS one can declare some transitions "unobservable"; to keep the presentation simple, we here omit all the observability aspects of MSOS. The rules below are as self-explanatory as one may expect; however, we intervene here and there to clarify some of MSOS' conventions and relationship with SOS:

Variable lookup:

$$x \xrightarrow{\{\sigma=\sigma_0,\ldots\}} \sigma_0[i] \text{ if } \sigma_0[x] \text{ defined} \tag{103}$$

Note that, rigorously speaking, one cannot replace the above with

$$x \longrightarrow \sigma[i] \text{ if } \sigma[x] \text{ defined} \tag{104}$$

because one does not know what σ is! Even though σ was added as a field at some moment in the definition of the language, each MSOS rule is expected to be self contained; otherwise one would pose a serious violation of modularity and of mathematical rigor, not to mention that language definitions originally intended to be very rigorous and formal need "hand-weaving" explanations. One of the important merits of MSOS is that it captured formally many of the "tricks" that language designers informally used in the past to avoid writing awkward and heavy SOS definitions. Some shortcuts are always welcome if explained properly and made rigorous enough locally, without having to rely on other rules. For example, the author of MSOS finds the following simpler version of

rule (103) acceptable

$$x \xrightarrow{\{\sigma,\ldots\}} \sigma[i] \text{ if } \sigma[x] \text{ defined,} \tag{105}$$

despite the fact that, strictly speaking, $\sigma[x]$ does not make sense by itself (recall that σ is a field name, not the state function) and that field names are expected to be paired with their semantic component in labels. Nevertheless, there is only one way to make the rule above make sense, namely to replace any use of σ by its semantic content, which therefore does not need to be mentioned.

Arithmetic expressions:

$$\frac{a_1 \xrightarrow{\mathcal{X}} a_1'}{a_1 + a_2 \xrightarrow{\mathcal{X}} a_1' + a_2} \tag{106}$$

$$\frac{a_2 \xrightarrow{\mathcal{X}} a_2'}{a_1 + a_2 \xrightarrow{\mathcal{X}} a_1 + a_2'} \tag{107}$$

Note that the same label $\mathcal{X}$ appeared both in the conditions and in the conclusions of the rules above. That means that the two transitions have precisely the same labels. Indeed, advancing an expression one step in the context of an addition expression has the same effects on the configuration as if the expression was advanced the same one step in isolation, without the other expression involved int he addition.

$$i_1 + i_2 \longrightarrow i \quad \text{where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{108}$$

$$\frac{a_1 \xrightarrow{\mathcal{X}} a_1'}{a_1 - a_2 \xrightarrow{\mathcal{X}} a_1' - a_2} \tag{109}$$

$$\frac{a_2 \xrightarrow{\mathcal{X}} a_2'}{a_1 - a_2 \xrightarrow{\mathcal{X}} a_1 - a_2'} \tag{110}$$

$$i_1 - i_2 \longrightarrow i \quad \text{where } i \text{ is } i_1 \text{ minus } i_2 \tag{111}$$

$$\frac{a_1 \xrightarrow{\mathcal{X}} a_1'}{a_1 * a_2 \xrightarrow{\mathcal{X}} a_1' * a_2} \tag{112}$$

$$\frac{a_2 \xrightarrow{\mathcal{X}} a_2'}{a_1 * a_2 \xrightarrow{\mathcal{X}} a_1 * a_2'} \tag{113}$$

$$i_1 * i_2 \to i \quad \text{where } i \text{ is the product of } i_1 \text{ and } i_2 \tag{114}$$

$$\frac{a_1 \xrightarrow{\mathcal{X}} a_1'}{a_1/a_2 \xrightarrow{\mathcal{X}} a_1'/a_2} \tag{115}$$

$$\frac{a_2 \xrightarrow{\mathcal{X}} a_2'}{a_1/a_2 \xrightarrow{\mathcal{X}} a_1/a_2'} \tag{116}$$

$$i_1/i_2 \to i \quad \text{where } i_2 \neq 0 \text{ and } i \text{ is the quotient of } i_1 \text{ by } i_2 \tag{117}$$

Relational operators

$$\frac{a_1 \xrightarrow{\mathcal{X}} a_1'}{a_1 \leq a_2 \xrightarrow{\mathcal{X}} a_1' \leq a_2} \tag{118}$$

$$\frac{a_2 \xrightarrow{\mathcal{X}} a_2'}{a_1 \leq a_2 \xrightarrow{\mathcal{X}} a_1 \leq a_2'} \tag{119}$$

$i_1 \leq i_2 \to \mathsf{true}$ where i_1 smaller than or equal to i_2 (120)

$i_1 \leq i_2 \to \mathsf{false}$ where i_1 larger than i_2 (121)

$$\frac{a_1 \xrightarrow{\mathcal{X}} a_1'}{a_1 \geq a_2 \xrightarrow{\mathcal{X}} a_1' \geq a_2} \tag{122}$$

$$\frac{a_2 \xrightarrow{\mathcal{X}} a_2'}{a_1 \geq a_2 \xrightarrow{\mathcal{X}} a_1 \geq a_2'} \tag{123}$$

$i_1 \geq i_2 \to \mathsf{true}$ where i_1 larger than or equal to i_2 (124)

$$i_1 \geq i_2 \to \mathsf{false} \quad \text{where } i_1 \text{ smaller than } i_2 \tag{125}$$

$$\frac{a_1 \xrightarrow{\mathcal{X}} a_1'}{a_1 \;=\; a_2 \xrightarrow{\mathcal{X}} a_1' \;=\; a_2} \tag{126}$$

$$\frac{a_2 \xrightarrow{\mathcal{X}} a_2'}{a_1 \;=\; a_2 \xrightarrow{\mathcal{X}} a_1 \;=\; a_2'} \tag{127}$$

$$i \;=\; i \to \mathsf{true} \tag{128}$$

$$i_1 \;=\; i_2 \to \mathsf{false} \quad \text{where } i_1 \neq i_2 \tag{129}$$

Boolean operators:

$$\frac{b_1 \xrightarrow{\mathcal{X}} b_1'}{b_1 \text{ and } b_2 \xrightarrow{\mathcal{X}} b_1' \text{ and } b_2} \tag{130}$$

$$\frac{b_2 \xrightarrow{\mathcal{X}} b_2'}{b_1 \text{ and } b_2 \xrightarrow{\mathcal{X}} b_1 \text{ and } b_2'} \tag{131}$$

t_1 and $t_2 \to t$ where t_1, t_2, t are truth values s.t. t is "t_1 and t_2" (132)

$$\frac{b_1 \xrightarrow{\mathcal{X}} b_1'}{b_1 \text{ or } b_2 \xrightarrow{\mathcal{X}} b_1' \text{ or } b_2} \tag{133}$$

$$\frac{b_2 \xrightarrow{\chi} b_2'}{b_1 \text{ or } b_2 \xrightarrow{\chi} b_1 \text{ or } b_2'} \tag{134}$$

t_1 or $t_2 \to t$ where t_1, t_2, t are truth values s.t. t is "t_1 or t_2" (135)

$$\frac{b \xrightarrow{\chi} b'}{\text{not } b \xrightarrow{\chi} \text{not } b'} \tag{136}$$

not $t \to t'$ where t and t' are opposite truth values (137)

Statements:

For statements, we prefer to follow the second small-step SOS variant. That is because in MSOS one needs some syntactic component to always be available in the configuration (needed for

sources and/or targets of transitions).

$$\frac{a \xrightarrow{\mathcal{X}} a'}{x := a \xrightarrow{\mathcal{X}} x := a'} \tag{138}$$

$$x := i \xrightarrow{\{\sigma=\sigma_0, \sigma'=\sigma_0[x \leftarrow i], \ldots\}} \mathsf{skip} \tag{139}$$

$$\frac{s_1 \xrightarrow{\mathcal{X}} s_1'}{s_1; s_2 \xrightarrow{\mathcal{X}} s_1'; s_2} \tag{140}$$

$$\mathsf{skip}; s_2 \rightarrow s_2 \tag{141}$$

For blocks, suppose that we chose to evaluate the enclosed statement until it becomes skip and then reduce the block containing only skipto skip:

$$\frac{s \xrightarrow{\mathcal{X}} s'}{\{s\} \xrightarrow{\mathcal{X}} \{s'\}} \tag{142}$$

$$\{\mathsf{skip}\} \to \mathsf{skip} \tag{143}$$

$$\frac{b \xrightarrow{\mathcal{X}} b'}{\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \xrightarrow{\mathcal{X}} \mathsf{if}\ b'\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2} \tag{144}$$

$$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to s_1 \tag{145}$$

$$\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \to s_2 \tag{146}$$

$$\mathsf{while}\ b\ \ s \to \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \ s)\ \mathsf{else}\ \mathsf{skip} \tag{147}$$

Programs:

$$\frac{s \xrightarrow{\chi} s'}{s; a \xrightarrow{\chi} s'; a} \tag{148}$$

$$\mathsf{skip}; a \to a \tag{149}$$

A major goal in MSOS definitions is to write on the labels as little information as possible and to use the implicit conventions regarding the missing information, because everything written on labels may work against modularity if the language is later on extended or simplified. As an extreme example, if one writes all the fields together with all their semantic contents on every single label, then MSOS becomes conventional SOS and therefore suffers from the same limitations as SOS in what regards modularity.

Recall the rules (95) and (96) for deriving the transitive closure

$\rightarrow^+$ of the small-step SOS relation $\rightarrow$. In order for two consecutive steps to "compose", the source configuration of the second had to be *identical* to the target configuration of the first. As one may expect, a similar thing must happen in MSOS, otherwise one may derive inconsistent computations. This process is explained in MSOS by making use of *category theory*, associating labels with morphisms in a category and then using the morphism composition mechanism of category theory.

However, category theory is not needed in order to understand how MSOS works in practice. A simple way to explain its label composition is by translating MSOS definitions back into SOS. Indeed, once one knows all the fields in the labels, which happens once a language definition is complete, one can automatically associate a standard small-step SOS definition to the MSOS one by replacing each MSOS rule with an SOS rule over configurations adding to the syntactic content the complete semantic content from

the label. The resulting SOS configurations will not have primed fields anymore, but only their unprimed variants; the primed fields are only used for saying how the target configuration modifies the source one. For example, assuming that γ contains the remaining semantic components in the configuration besides σ (environments, outputs, stacks, etc.), the MSOS rule (139) for assignment translates into an SOS rule of the form

$$\langle x := i,\ \sigma, \gamma\rangle \to \langle \mathsf{skip}, \sigma[x \leftarrow i], \gamma\rangle \tag{150}$$

Also, conditional rules like (106) having a common label in both their conditions and conclusions are translated into

$$\frac{\langle a_1, \sigma, \gamma\rangle \to \langle a_1', \sigma', \gamma'\rangle}{\langle a_1 + a_2, \sigma, \gamma\rangle \to \langle a_1' + a_2, \sigma', \gamma'\rangle}. \tag{151}$$

This way, MSOS becomes SOS and we can borrow the transitive closure $\to^+$ of the one-step relation from SOS.

Advantages of MSOS

1) *MSOS adds modularity to SOS* definitions by allowing one to refer only to configuration items of interest in each rule and using "..." for the remaining ones. This way, adding rules for new language features that need additional configuration infrastructure can be done without having to modify any of the rules for unrelated existing features.

Homework Exercise 2 ***(10 points)*** *Add rules for* `halt` *a (this new statement was explained in the previous homework exercise) to the MSOS definition above. How many rules did you need? How many existing rules did you have to change?*

(Hint: you may need to add one more field in labels to store the "halting signal"; feel free to also change the syntax of the language however you wish, in particular to add one more top-level program construct if you need to "catch" the halting signal).

2) By allowing unobservable transitions, MSOS provide better support for allowing the language designer tune the computation granularity of defined language features.

Disadvantages of MSOS

While modularity was a major drawback of conventional SOS and MSOS proposes solutions to improve it, MSOS still carries most of the other disadvantages of SOS. For example, it still has an interleaving semantics for concurrency and it still says nothing about models. While providing better support for defining the desired computational granularity of language constructs by allowing certain transitions to be unobservable, MSOS still needs to propagate information through the program syntax in order for that information to reach its destination. For example, in the case of halt, the "halting signal" still had to be propagated from subexpressions to parent expressions in labels; it is true that the program did not need to be changed anymore like in the case of

small-step SOS, but the number of rule applications remains pretty much the same as in small-step SOS; therefore, we still cannot model the intended computational granularity of halt a, which is "evaluate a to i and then stop immediately the execution of the entire program with result i". We will shortly see how reduction semantics with evaluation contexts allows us to do that.

Like conventional SOS, MSOS is still rather low-level, mimicking by logical inference both one-step computations and propagation of reductions along the language constructs. That means that computation contexts are captured as proof-contexts and, unfortunately, proof contexts cannot be stored and/or retrieved at will. That means that complex control-intensive language constructs like callcc may be impossible to define in MSOS.

A disadvantage of MSOS compared to SOS is that it is even more syntactic in its nature than SOS: every transition must contain pieces of syntax in both sides. Recall our first small-step SOS

variant, which allowed us to derive transitions of the form $\langle s, \sigma \rangle \to \langle \sigma' \rangle$ when the statement s carried no further computation. To write that in MSOS, one would need to first generate an artificial "statement value" and then discard it with another rule. One could find that natural to do, especially in the context of a functional programming language, but on the other hand one can argue that one should not be enforced to produce an artificial value for things that are commonly used exclusively for their side effects.

Reduction Semantics with Evaluation Contexts

Introduced in

> A. K. Wright and M. Felleisen. *A syntactic approach to type soundness*. Information and Computation, 115(1):38–94, 1994.

also called *context reduction*, the reduction semantics with evaluation contexts style improves over small-step SOS definitional styles in at least three ways:

1. It gives a *more compact semantics to context-sensitive reduction*, by using parsing to find the next redex rather than towered small-step rules;
2. It provides the possibility of also *modifying the context* in which a reduction occurs, making it much easier to control the execution of programs. For example, defining halt is done now

using only one rule, $\langle c[\mathsf{halt}\, i], ...\rangle \rightarrow \langle i\rangle$, where c is the current evaluation context, preserving the desired computational granularity;

3. It allows the possibility to handle evaluation contexts like any other values in the language, in particular to pass them to other contexts or to store them. This way one can more easily define the semantics of control-intensive language features such as return of functions, exceptions, even callcc(call-with-current-continuation, such as, e.g., in Scheme).

In a context reduction semantics of a language, one typically starts by defining the syntax of *contexts*. A context is a program or fragment of program with a "hole", the hole being a placeholder where the next computational step takes place. If c is such a context and e is some well formed fragment of program (expression, statement, etc.), then $c[e]$ is the program or fragment of program formed by replacing the hole of c by e. For simplicity, we consider

only one type of contexts here, but in general one can have various types, depending upon the type of their "holes". The *characteristic reduction step underlying context reduction* is of the form

$$\frac{\langle e, \gamma\rangle \to \langle e', \gamma'\rangle}{\langle c[e], \gamma\rangle \to \langle c[e'], \gamma'\rangle},$$

capturing the fact that reductions are allowed to take place only in appropriate evaluation contexts, where γ (and so does γ') consists of zero, one or more configuration resources that are necessary to evaluate e (and e', respectively), such as stores, stacks, locks, etc. If γ is empty (we'll see such an example shortly), then we omit configurations entirely and write the rule above as

$$\frac{e \to e'}{c[e] \to c[e']},$$

how it was first introduced and how is commonly seen in context reduction definitions of functional languages.

When a rule like the above is applied, we say that "e reduces to e' in context c. Interestingly, the reduction step underlying context reduction tends to be the only so-called "conditional" rule, in the sense that the remaining rules take no reduction condition (they have a dot above the line) – they may still require side conditions.

Therefore, an important part of a context reduction semantics is the definition of evaluation contexts, which is typically done by means of a context-free grammar. In what follows we give two complete context reduction semantic definitions for our simple language, one in which the state is carried as part of transitions, and another one in which the state is encapsulated as a top level attribute of the context.

Variant 1

Let us first define the syntax of evaluation contexts for our first variant as follows (we let [] denote the "hole"):

$$
\begin{array}{lll}
Cxt & ::= & \Box \\
& | & Cxt + AExp \mid AExp + Cxt \mid \ldots(\text{same for } -, *, /) \\
& | & Cxt \leq AExp \mid AExp \leq Cxt \mid \ldots(\text{same for } \geq, =) \\
& | & \mathsf{not}\ Cxt \mid Cxt\ \mathsf{and}\ BExp \mid BExp\ \mathsf{and}\ Cxt \mid \ldots(\text{same for } \mathsf{or}) \\
& | & Var := Cxt \mid Cxt; Stmt \mid \{Cxt\} \\
& | & \mathsf{if}\ Cxt\ \mathsf{then}\ Stmt_1\ \mathsf{else}\ Stmt_2 \\
& | & Cxt; AExp
\end{array}
$$

A context therefore gives a term with precisely one hole, with the intuition that the hole is a placeholder for a fragment of program

that can be evaluated next. Note that we did not add a production "$\mathit{Cxt} ::= \ldots \mid \mathsf{while}\ \mathit{Cxt}\ \mathit{Stmt}$" as one may (wrongly) expect. That is because such a production would allow the evaluation of the boolean expression in the while loop to a boolean value; supposing that that value is true, then, unless one modifies the syntax in some rather awkward way, there is no chance to recover the original boolean expression to evaluate it after the evaluation of the statement. The preferred solution to handle while loops remains the same as in SOS, namely to explicit unroll them into conditional statements.

Here are some examples of correct evaluation contexts:

$\Box$

$3 + \Box$

$3 * (\Box + 7)$

$x * (\Box + 3 * y)$

$\mathsf{if}\ \square\ \mathsf{then}\ \mathit{Stmt}_1\ \mathsf{else}\ \mathit{Stmt}_2$, where Stmt_1 and Stmt_2 are any well-formed statements

$\{\square; \mathsf{skip}; x := 5\}$

$\square; x + y$

Here are some examples of incorrect evaluation contexts:

$\square + (2 + \square)$ – a context can have only one hole

$x + y$ – a context must contain a hole

$x + y; \square$ – one cannot place a semicolon after an expression in our grammar

$\square := 7$ – the grammar of contexts disallows this; holes are placeholders where computations can take place

$\mathsf{if\,true\ \ then}\ \square\ \mathsf{else\,skip}$ – a hole (or a computational step) can only appear in the condition of a conditional

$\{\mathsf{skip}; \square\}$ – one cannot compute the second statement in a sequential composition; skipmust be discarded first

The context reduction operational semantics style is based on a tacitly assumed parsing-like mechanism that takes a program or a fragment of program p and decomposes it into a context c and a subprogram or fragment of program e, such that $p = c[e]$. Here are some examples of such decompositions:

$$7 = (\Box)[7]$$
$$3 + x = (3 + \Box)[x] = (\Box + x)[3] = (\Box)[3 + x]$$
$$3 * (2 * x + 7) = (3 * (\Box + 7))[2 * x] = (\Box * (2 * x + 7))[3] = ...$$
$$x * (y + x * (x - y) + 3 * y) = (x * (\Box + 3 * y))[y + x * (x - y)]$$

If one picks the empty context $c = \Box$ in the decomposition of p as $c[e]$, then e is p and thus the context reduction rule is useless. Context reduction is effective when the context is chosen to be large, the largest if possible, meaning that its hole is as "deep" as possible. Then one only needs to worry about giving rules capturing only basic computation steps and not how computations

are propagated along the syntax. With this in mind, we can almost mechanically translate our small-step SOS rules as follows, where we ignore the empty conditions in all rules (the dot above the line together with the useless line).

Context reduction rule:

$$\frac{\langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\langle c[e], \sigma\rangle \to \langle c[e'], \sigma'\rangle},$$

Variable lookup:

$\langle x, \sigma\rangle \to \langle \sigma[x], \sigma\rangle$ if $\sigma[x]$ defined

Arithmetic expressions:

$\langle i_1 + i_2, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ where i is the sum of i_1 and i_2

$\langle i_1 - i_2, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ where i is i_1 minus i_2

$\langle i_1 * i_2, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ where i is the product of i_1 and i_2

$\langle i_1 / i_2, \sigma \rangle \rightarrow \langle i, \sigma \rangle$ where $i_2 \neq 0$ and i is the quotient of i_1 by i_2

Relational operators:

$\langle i_1 \leq i_2, \sigma \rangle \rightarrow \langle \mathsf{true}, \sigma \rangle$ where i_1 smaller than or equal to i_2

$\langle i_1 \leq i_2, \sigma \rangle \rightarrow \langle \mathsf{false}, \sigma \rangle$ where i_1 larger than i_2

$\langle i_1 \geq i_2, \sigma \rangle \rightarrow \langle \mathsf{true}, \sigma \rangle$ where i_1 larger than or equal to i_2

$\langle i_1 \geq i_2, \sigma \rangle \rightarrow \langle \mathsf{false}, \sigma \rangle$ where i_1 smaller than i_2

$\langle i = i, \sigma \rangle \rightarrow \langle \mathsf{true}, \sigma \rangle$

$\langle i_1 = i_2, \sigma \rangle \rightarrow \langle \mathsf{false}, \sigma \rangle$ where $i_1 \neq i_2$

Boolean operators:

$\langle t_1 \text{ and } t_2, \sigma\rangle \to \langle t, \sigma\rangle$ where t_1, t_2, t are truth values s.t. t is "t_1 and t_2"

$\langle t_1 \text{ or } t_2, \sigma\rangle \to \langle t, \sigma\rangle$ where t_1, t_2, t are truth values s.t. t is "t_1 or t_2"

$\langle \text{not } t, \sigma\rangle \to \langle t', \sigma\rangle$ where t and t' are opposite truth values

Statements:

$\langle x := i, \sigma\rangle \to \langle \text{skip}, \sigma[x \leftarrow i]\rangle$

$\langle \text{skip}; s_2, \sigma\rangle \to \langle s_2, \sigma\rangle$

$\langle \{\text{skip}\}, \sigma\rangle \to \langle \text{skip}, \sigma\rangle$

$\langle \text{if true then } s_1 \text{ else } s_2, \sigma\rangle \to \langle s_1, \sigma\rangle$

$\langle \text{if false then } s_1 \text{ else } s_2, \sigma\rangle \to \langle s_2, \sigma\rangle$

$\langle \text{while } b\ s, \sigma\rangle \to \langle \text{if } b \text{ then } (s; \text{ while } b\ s) \text{ else skip}, \sigma\rangle$

Programs:

$\langle s; a\rangle \rightarrow \langle s; a, \emptyset\rangle$

$\langle \mathsf{skip}; a, \sigma\rangle \rightarrow \langle a, \sigma\rangle$

Exercise 14 *Note that our approach above was to first evaluate the statement enclosed by a block to* skip *and then eliminate the block. Change the context reduction definition above so that the block statement construct is eliminated as soon as encountered.*

A drawback of the approach adopted in this variant of context reduction is that the state had to be carried as part of the configurations through all the rules above, even though most of the rules had no need for it; it had to be carried because it was needed in two of the rules, namely the variable lookup and the assignment.

Note that, with the current definition of contexts, the state indeed needs to be passed by each rule, precisely because the rules for variable lookup and for variable assignment need it. We will next

see a more common variant of context reduction in which the state is encapsulated as part of the context.

Variant 2

Another context reduction approach is to include the remaining of the configuration, only the state in our case, as part of the context. The advantage of this approach is that most of the remaining reduction rules become simpler, in the sense that they only show the syntactic transformations. The disadvantage of this approach is that what used to be called "syntax" now contains rather semantic components, such as the state of the program. Of course, if one includes everything into syntax, including the semantics, then the syntacticists' slogan "everything is syntax" makes full sense.

Here is the new definition of contexts:

$$\begin{array}{rcl} \mathit{Cxt} & ::= & \text{...(same as before, plus the next production)} \\ & | & \langle \mathit{Cxt}, \mathit{State} \rangle \end{array}$$

The remaining rules can now be changed as follows:

Context reduction rule:

$$\frac{e \to e'}{c[e] \to c[e']} \tag{152}$$

Variable lookup:

$$\langle c, \sigma \rangle [x] \to \langle c, \sigma \rangle [\sigma[x]] \text{ if } \sigma[x] \text{ defined} \tag{153}$$

Arithmetic expressions:

$$i_1 + i_2 \to i \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{154}$$

$$i_1 - i_2 \to i \text{ where } i \text{ is } i_1 \text{ minus } i_2 \tag{155}$$

$i_1 * i_2 \rightarrow i$ where i is the product of i_1 and i_2 (156)

$i_1 / i_2 \rightarrow i$ where $i_2 \neq 0$ and i is the quotient of i_1 by i_2 (157)

Relational operators:

$i_1 \leq i_2 \rightarrow \mathsf{true}$ where i_1 smaller than or equal to i_2 (158)

$i_1 \leq i_2 \rightarrow \mathsf{false}$ where i_1 larger than i_2 (159)

$i_1 \geq i_2 \rightarrow \mathsf{true}$ where i_1 larger than or equal to i_2 (160)

$i_1 \geq i_2 \rightarrow \mathsf{false}$ where i_1 smaller than i_2 (161)

$i = i \rightarrow \mathsf{true}$ (162)

$i_1 = i_2 \rightarrow \mathsf{false}$ where $i_1 \neq i_2$ (163)

Boolean operators:

t_1 and $t_2 \to t$ where t_1, t_2, t are bools s.t. t is "t_1 and t_2" (164)

t_1 or $t_2 \to t$ where t_1, t_2, t are bools s.t. t is "t_1 or t_2" (165)

not $t \to t'$ where t and t' are opposite truth values (166)

Statements:

$\langle c, \sigma \rangle [x := i] \to \langle c, \sigma[x \leftarrow i] \rangle [\mathsf{skip}]$ (167)

$\mathsf{skip}; s_2 \to s_2$ (168)

$\{\mathsf{skip}\} \to \mathsf{skip}$ (169)

if true then s_1 else $s_2 \to s_1$ (170)

if false then s_1 else $s_2 \to s_2$ (171)

while b $s \to$ if b then $(s;$ while b $s)$ else skip (172)

Programs:

$$\langle s; a\rangle \longrightarrow \langle s; a, \emptyset\rangle \tag{173}$$

$$\mathsf{skip}; a \longrightarrow a \tag{174}$$

From here on we use the second context reduction variant above, because it is more compact, more elegant and more broadly used than the first one.

Notice the rule (173) that initializes the state of a program configuration. Even though the second variant of context reduction has no configurations defined (since the state is included as part of contexts) we tacitly assumed configurations to also be part of the context reduction definition. Without configurations, context instantiations of the form $\langle c, \sigma\rangle[e]$ would be undefined entities; indeed, our grammar for contexts and/or programs cannot parse, for example, $\langle \mathsf{skip}; 3\rangle$, which is equal to $\langle \Box; 3\rangle[\mathsf{skip}]$. Such context instances are used in the rules for variable lookup and assignment.

Context reduction is an inherently small-step structural operational semantic approach: for each of the two variants discussed, the rules above define precisely one step of computation. Therefore, in order to completely evaluate a program one needs to define a transitive closure of the one-step transition relation, as we did for small-step SOS. Let $\rightarrow^+$ denote the *complete transitive closure* of the one-step context reduction relation defined as follows (*Config* and *Config'* are program or arithmetic expression configurations, and i are integer numbers):

$$\frac{Config \rightarrow \langle i, \sigma \rangle}{Config \rightarrow^+ \langle i \rangle} \tag{175}$$

$$\frac{Config \rightarrow Config',\ Config' \rightarrow^+ \langle i \rangle}{Config \rightarrow^+ \langle i \rangle} \tag{176}$$

Let us next discuss an example of reduction using the second

context reduction variant above. Our goal is to derive

$$\langle \{x := 1; y := 2; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0\}; x + y\rangle \rightarrow^{+} \langle 1\rangle.$$

The only rule which can be applied on the program configuration above is (173), obtaining the derivation:

$$\begin{array}{rl} & \langle \{x := 1; y := 2; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0\}; x + y\rangle \\ \rightarrow & \langle \{x := 1; y := 2; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0\}; x + y, \emptyset\rangle \end{array}$$

From here on we use the standard notation for instantiated contexts in which the redex is placed in a box replacing the hole of the context. For example, the fact that expression $2 * (x * x + y)$ is seen as a context instance $(2 * (\Box + y))[x * x]$ is written compactly and intuitively as follows: $2 * (\boxed{x * x} + y)$. With this notation, one gets the following sequence of reductions, where the number above the arrow represents the rule number that was applied; boxed

numbers symbolize that the corresponding reduction step was applied in context, using an implicit application of the context reduction rule (152):

$$
\begin{array}{ll}
 & \langle \{ \boxed{x := 1}; y := 2; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0 \}; x + y, \emptyset \rangle \\
\xrightarrow{(167)} & \langle \{ \boxed{\mathsf{skip}; y := 2; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0} \}; x + y, (x \leftarrow 1) \rangle \\
\xrightarrow{\boxed{(168)}} & \langle \{ \boxed{y := 2}; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0 \}; x + y, (x \leftarrow 1) \rangle \\
\xrightarrow{(167)} & \langle \{ \boxed{\mathsf{skip}; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0} \}; x + y, (x \leftarrow 1, y \leftarrow 2) \rangle \\
\xrightarrow{\boxed{(168)}} & \langle \{ \mathsf{if}\, x \geq \boxed{y} \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0 \}; x + y, (x \leftarrow 1, y \leftarrow 2) \rangle \\
\xrightarrow{(153)} & \langle \{ \mathsf{if}\, \boxed{x} \geq 2 \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0 \}; x + y, (x \leftarrow 1, y \leftarrow 2) \rangle \\
\xrightarrow{(153)} & \langle \{ \mathsf{if}\, \boxed{1 \geq 2} \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0 \}; x + y, (x \leftarrow 1, y \leftarrow 2) \rangle \\
\xrightarrow{\boxed{(161)}} & \langle \{ \boxed{\mathsf{if}\, \mathsf{false}\, \mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0} \}; x + y, (x \leftarrow 1, y \leftarrow 2) \rangle \\
\xrightarrow{\boxed{(171)}} & \langle \{ \boxed{y := 0} \}; x + y, (x \leftarrow 1, y \leftarrow 2) \rangle \\
\xrightarrow{(167)} & \langle \boxed{\{\mathsf{skip}\}}; x + y, (x \leftarrow 1, y \leftarrow 0) \rangle \\
\xrightarrow{\boxed{(169)}} & \langle \boxed{\mathsf{skip}; x + y}, (x \leftarrow 1, y \leftarrow 0) \rangle \\
\xrightarrow{\boxed{(174)}} & \langle x + \boxed{y}, (x \leftarrow 1, y \leftarrow 0) \rangle \\
\xrightarrow{(153)} & \langle \boxed{x} + 0, (x \leftarrow 1, y \leftarrow 0) \rangle \\
\xrightarrow{(153)} & \langle \boxed{1 + 0}, (x \leftarrow 1, y \leftarrow 0) \rangle \\
\xrightarrow{\boxed{(154)}} & \langle 1, (x \leftarrow 1, y \leftarrow 0) \rangle
\end{array}
$$

Note that the evaluation context changed almost at each step during the reduction sequence above. Applying rule (175) on the last reduction step and then rule (176) iteratively in a bottom-up fashion along the remaining reductions, we eventually derive

$$\langle \{x := 1; y := 2; \mathsf{if}\, x \geq y \,\mathsf{then}\, x := 0 \,\mathsf{else}\, y := 0\}; x + y \rangle \rightarrow^{+} \langle 1 \rangle.$$

Exercise 15 *The reduction rule for while loops* (172) *says that any while loop is translated into a conditional containing the same while loop in its "then" branch. What does prevent us from unrolling the while loop in the "then" branch applying the same rule* (172) *over and over again, thus obtaining a non-terminating reduction?*

Advantages of Context Reduction

1) *Evaluation contexts are explicit first-class entities* in context reduction, so they can be modified, passed, stored and retrieved. This allows for elegant definitions of control intensive language features such as abrupt termination of programs using halt-like constructs (just dissolve the current evaluation context), functions with return (stack the evaluation contexts at function calls and restore them at return), exceptions, and even callcc (store the current evaluation context as a value in the store, retrieve it when applied). Moreover, these operations on contexts can be done in such a way that they have the expected computation granularity.

Homework Exercise 3 ***(10 points)*** *Add context reduction definitions for increment and halt (same language constructs as in the first homework exercise). Give definitions using both context reduction styles described above.*

2) Since reductions can only happen in proper evaluation contexts and propagation of computation is done once and for all using the context reduction rule, one needs not worry about propagating local computations up-wards along the syntax; this way, definitions in context reduction become *significantly more compact* than SOS definitions.

Disadvantages of Context Reduction

1) Context reduction is based on an *implicit mechanism to split a program* or a fragment of program p into an evaluation context c and a fragment of program, also called *redex*, e, so that $p = c[e]$. This split operation is somehow assumed atomic and non-computational, so it elegantly captures and hides all the small-step propagation rules of SOS. However, this decomposition is not obvious and may require significant computational resources, sometimes linear in the size of the program. Moreover, since a new evaluation context needs to be recalculated at almost any reduction

step, it can easily become the major bottleneck in the efficiency of an interpreter implemented following the context reduction approach. While such an interpreter would be no worse than one implemented following the small-step SOS approach, it can still be a lot slower than it needs to be. Techniques such as *refocusing* have been proposed to incrementally compute the next evaluation context, but these techniques appear to work only when the decomposition of the program into an evaluation context and a redex is deterministic.

2) It is somehow still awkward to define concurrent languages using context reduction. That is because concurrent processes typically communicate with each other from different places in the configuration, so one may need contexts with more than one hole, preferably with an arbitrary number of them. Computing and matching such such contexts on complex configurations becomes a significantly non-trivial issue.

3) It still does not serve as a solid foundation for concurrency, because, just like SOS, it only captures an interleaving semantics of a defined concurrent system.

4) It still says nothing about models, being essentially a purely syntactic reduction technique.

We are going to shortly discuss a language definitional technique, called K, which has none of the limitations above. Moreover, we'll present an automatic translation from any context reduction definition, say CR, into a K definition, say k(CR), which is faithful to the computational granularity of the original context reduction definition, in the sense that any computational step in CR has a precise one-step correspondent in k(CR) and vice-versa, that is, any computational step in k(CR) has a precise correspondent step in CR. Additionally, k(CR) is as compact as the original CR: it adds one equation per context evaluation production and one K-rule per context reduction rule. Therefore, if one wants to use context

reduction within K then one is free to do so, but, of course, one cannot circumvent the limitations of context reduction. As shown in the later classes by a series of languages defined in K, such as imperative, functional, object-oriented and logic programming, as well as concurrent languages, one may be better off using the full strength of K, unlimited by restricted methodological fragments of K, such as context reduction.